We are IntechOpen, the world's leading publisher of Open Access books Built by scientists, for scientists

4,800
Open access books available

122,000
International authors and editors

135M
Downloads

Our authors are among the

154
Countries delivered to

TOP 1%
most cited scientists

12.2%
Contributors from top 500 universities

CLARIVATE ANALYTICS BOOK CITATION INDEX INDEXED

WEB OF SCIENCE™

Selection of our books indexed in the Book Citation Index in Web of Science™ Core Collection (BKCI)

Interested in publishing with us?
Contact book.department@intechopen.com

Numbers displayed above are based on latest data collected.
For more information visit www.intechopen.com

17

Pedagogical use of automatic graders

Isidoro Hernán-Losada[1], Cristóbal Pareja-Flores[2]
and J. Ángel Velázquez-Iturbide[1]
[1]*Dept. de Lenguajes y Sistemas Informáticos I, Universidad Rey Juan Carlos, Móstoles, Spain*
[2]*Dept. de Sistemas Informáticos y Computación, Universidad Complutense de Madrid, Spain*

1. Introduction

We make a proposal to learn programming, based on several elements: the test-first approach to learning and software development, automatic grading, and the cognitive domain of Bloom's Taxonomy. In this section, we give a brief overview of them.

1.1. Test-Driven Approach

In the approach of test-driven design (TDD) to software development (Beck, 2001; Shepard *et al.*, 2001), program design starts by writing a test for the new piece of code, even before designing the piece itself. Test cases are an informal but precise way to specify the expected behavior of the program. In this context, the primary role of a test is specification, playing validation a secondary role.

In last years, TDD has been broadly accepted, both in the academic and the professional worlds (Beck, 1999; Beck, 2001; Shepard *et al.*, 2001). In the academic world, the approach is often called test-first. It has been noticed that designing test cases forces the novice programmer to better understand and model the program requirements, forcing the student to prevent any situation that the program may find, and, in summary, to improve the students' performance. In the professional world, several design methodologies make an intensive of testing. The most popular of these methodologies is probably extreme programming (XP, 2006). This methodology advocates first-test as one of its principles: it helps the developer to actually consider the needs to be accomplished, because requirements are nailed firmly by tests.

We don't pursue in this approach the goal of having students designing tests, because nearly all the automatic graders in use today request (correct) sets of data previously given. However, reading these data by the student forces her to think in tests before thinking in the program itself, and fosters her to have a positive attitude to TDD. In this work, we assume that all the programming activities are both described and tested by samples of data cases.

1.2. Automatic Grading

The use of automatic graders is currently a consolidated means to evaluate students' assignments, both for educational purposes and for programming contests, where this sort of tools is currently essential. Basically, an automatic grader receives a program, as the intended solution for a problem, and evaluates it under several criteria (see Figure 1).

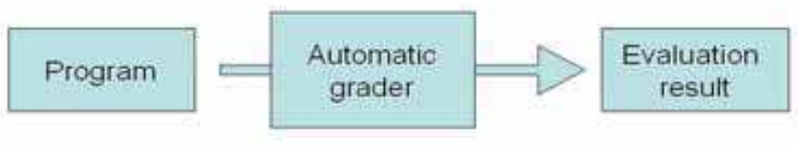


Fig. 1. Basic operation of a grader

Some of the criteria to evaluate a program are static, in the sense that the program doesn't need to be run (correct syntax, style, similarity due to plagiarism, etc.). Testing these features is highly dependent on the language. Other criteria (correctness, efficiency) are dynamic, i.e. assessing a program with respect to these criteria requires compiling it, running it on a known, hidden set of data, and comparing the results with the expected ones. Also, the program is requested to constrain to time and space constraints.
Automatic grading has many educational advantages (Daly & Waldron, 2002):

- Immediate feedback to students, so they can self-study at their own pace.
- More objective and fair grading.
- Permanent access, without time or space limitations, thus suitable for distance learning and fostering auto-learning.
- Efficiency of the grading process. In particular, it is suitable for large classrooms (Hunt *et al.*, 2002) or for very frequent grading (Douce *et al.*, 2005).
- Fostering the TDD approach.

We also find non-traditional ways of using correctors in education. For instance, programming contest problems can be modified to successfully teach data structures and algorithms (Szuecs, 2001). Programming contests with automatic graders may also be used to foster collaborative work. A team must extract the essence of the statement, formulate it as a mathematical problem, and then apply robust computer science theory to solve it (Shilov & Yi, 2002).
There are a myriad of graders currently available. For the interested reader, we just cite a couple of surveys (Ala-Mutka, 2005; Douce *et al.*, 2005) and the name of some more recent graders: PC^2, Mooshak, RockTest, PC-CAT and the Online Judge. It also exists a large number of repositories [UVA, 2009] of problems of programming contests, but they are only used to train contestants' (who are experienced programmers).

1.3. Bloom's Taxonomy

Bloom's taxonomy (Bloom *et al.*, 1956) is a classification of learning objectives. It distinguishes three domains: affective, psychomotor, and cognitive. In programming learning, the cognitive domain is the most important, so we refer to it in this paper. This domain includes six levels hierarchically organized (see Figure 2), where each level is a prerequisite for the higher ones.

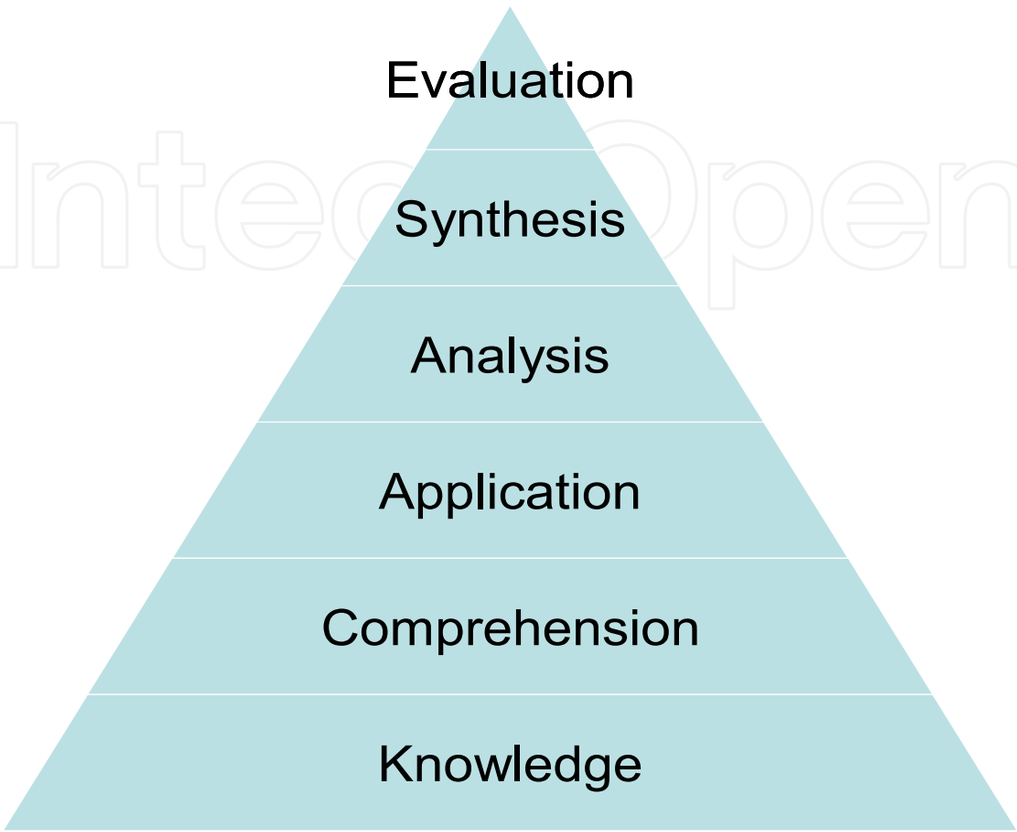


Fig. 2. Categories in the cognitive domain of Bloom's Taxonomy.

Others authors have made revisions of this framework, e.g., Anderson and Krathwohl (2001). However, Bloom's taxonomy is a widely accepted framework and we consider it the most suitable to assess student's level in programming.

1.4. Our Contribution

Typically, programming assignments require the student to develop a running program. Whilst this goal is arguably essential, it can vary the material given to the student to solve the problem, the simplicity of the problem, the size of the solution, etc. There are attempts to classify exercises according to several criteria, such as difficulty (Pareja-Flores & Velázquez-Iturbide, 2000), constituent parts or intended use (Edwards *et al.*, 2008; Gregorio-Rodríguez *et al.*, 2001). However, all of these classifications have the drawback that they do not consider pedagogic criteria.

From a pedagogical point of view, requiring students to deliver a running program has negative consequences on their attitude and learning success. Most assignments are aimed at the synthesis level of Bloom's taxonomy, where the program to design is non-trivial. As a

consequence, the novice programmer tackles problems of great difficulty from the beginning, a situation that often frustrates her and encourages her to drop away the study of this subject.

Fortunately, we may remove the deficiencies of this approach by considering automatic grading and learning levels. We advocate the use of problems oriented to TDD, like problems of programming contests, and assessing them with an automatic grader. The student may benefit from the grader feedback to obtain formative information, not just summative one. In addition, problems must be designed in appropriate order to achieve the levels of comprehension, application, analysis and synthesis of Bloom's taxonomy.

In this chapter, we describe a set of criteria to design programming problems, suitable for automatic grading and aimed at the four central levels of Bloom's taxonomy. A pedagogical system may extract and show appropriate problems, depending on the course goals or the stages in the learning process. These problems have additional advantages: they increase the motivation and foster to practice reading programs.

1.5. Organization of the Chapter

Section 2 describes the features of test-driven programming exercises. The third section applies Bloom's taxonomy to programming, illustrating each level with test-driven problems. Section 4 contains our proposal of criteria for associating educational meta-information to assignments and for guiding an assignment collection. Section 5 discusses our proposal. Then, we summarize our conclusions and future works.

2. Test-Driven Programming Assignments

In order of better explain the advantages of TDD in programming learning, we start with an example: to solve a second-order equation, represented $a \cdot x^2 + b \cdot x + c = 0$. An adequate set of test cases reveals whether first-degree and zero-degree equations, as well as imaginary solutions, are allowed, and how to display each case. In addition, these tests can be used to test whether the program works correctly.

Our proposal starts by providing every problem with associated test cases in a fixed format. They are a mandatory piece of the problem statement because the final program will be graded by testing it with respect to these data cases.

Although students will not practise the production of tests, these specifications can range over a wide set of possibilities:

- Specifications may include additional information, like several possible solutions to choose one.
- A problem statement may include an incomplete solution to be completed, or a wrong one to be fixed.
- The solution to a problem may be given directly in pseudo-code, ranging from a very high level of abstraction to low level or even almost literally.

The level of difficulty (in terms of Bloom's taxonomy) should also be identified. It strongly depends on the amount of information provided and the stage in the curriculum where a problem is located.

3. Programming Learning and Bloom's Taxonomy

The predominant activity in programming learning is to develop a program that satisfies a statement or specification, sometimes illustrated with one or several examples. These problems are more interesting when they demand a number of language constructs and techniques (Pareja-Flores & Velázquez-Iturbide, 2000), so they are too difficult in the first stages of learning. Stated in terms of the Bloom's taxonomy, the difficulty of these problems fit the application, analysis or synthesis levels.

In this section, we review all the levels of Bloom's taxonomy and give some examples of assignments aimed at each level. We select problems that can be solved with an automatic grader.

- Knowledge level: The student remembers single pieces of information. It is the lowest level. The form of the questions and the degree of precision required should not be too different from the way in which the knowledge was originally learned. Due to the variety of methods to acquire knowledge, we will not include assignments for programming in this level.
- Comprehension level: The student understands the meaning; typical tasks in this level are translation, prediction of the behavior, or summarization of pieces of code.
    - One of the simplest sorts of programs consists in performing a calculation by means of a known formula. A related problem aimed at the comprehension level consists in rewriting an algorithmic description, such as pseudo-code, a flowchart or natural language, into an executable program.
    - A second sort of exercises is to give a problem statement, an incomplete program and a collection of fragments to choose the one(s) that correctly fills the incomplete program. A variation consists in re-ordering a complete but incorrect program into a correct one to do a given task.
- Application level: The student uses a concept in a new situation, makes unprompted use of an abstraction, or applies what has learned in the classroom in novel situations in the work place.
    - The simplest programs in this level require applying a known formula or method in a new situation. The students may be given a model of program (e.g. Pythagoras) and re required to produce other programs for different purposes (e.g. a circle's area, Fahrenheit to Celsius conversion, solving a second-degree equation with real solutions, etc.). Another example consists in sorting a vector of non-numerical elements, assuming that she understands sorting algorithms with arrays of numbers.
- Analysis level: The student separates material or concepts into component parts so that he understands its structure and organization. Bloom suggests that adding new material is a genuine test of the participants' analytical abilities. In effect, they did not have the opportunity to recall analytical comments from previous discussions or readings that were provided by others.
    - As the complexity of problems increases, so does the need of stepwise refinement, decomposing a task into simpler ones. Depending on the case, these subtasks may be modeled as subprograms, modules, methods, etc.

Analysis is complemented with synthesis in the task to solve a new problem, because neat decomposition does not lead to a program if the parts are not also composed together. This idea is specifically supported by the revised Bloom's taxonomy (see Figure 3). The border

that separates the analysis and synthesis levels is, in programming, rather diffuse, and often indistinguishable.

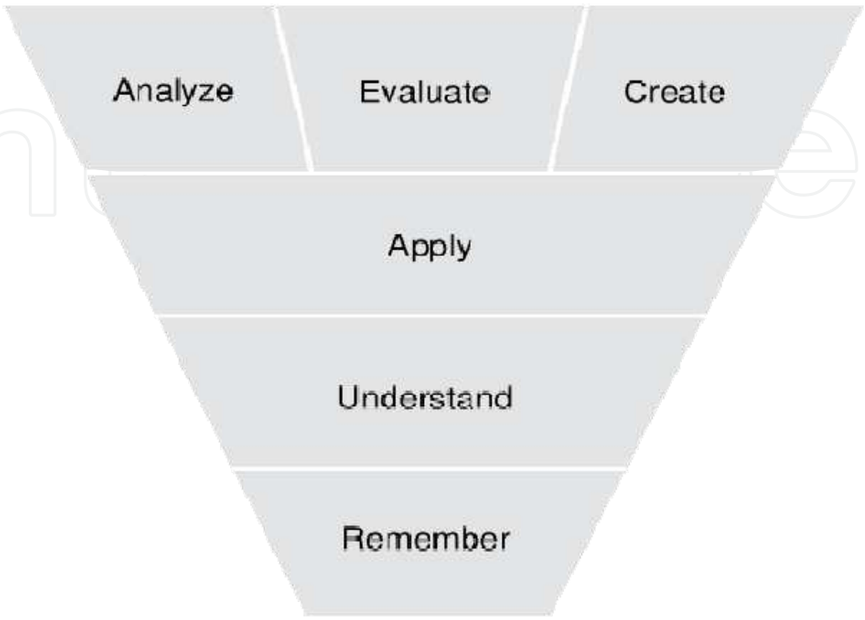


Fig. 3. "Flattened" Bloom's taxonomy (Anderson & Krathwohl, 2001)

- Synthesis level: The student builds a structure or pattern from several elements, put parts together to form a whole, or create a new meaning or structure. In testing, this level of Bloom's taxonomy suggests that too much control and too detailed instruction will stifle the creativity that you are trying to capture. This level gets more important as the student deepens in the study of programming constructs and techniques, and it becomes the predominant level in programming with abstract data types, with libraries and with advanced algorithms (like backtracking or dynamic programming). Its high position in Bloom's taxonomy makes it difficult to many students, especially to non majors.
    - Design a program for a new problem. The difficulty is higher than in the previous level: not only a formula must be rewritten, or a large program must be refined with many components. The difficulty lies in the subtle nature of the problem itself, and proficiency and expertise is necessary to achieve a final, acceptable solution.
- Evaluation level: The student makes judgments about the value of ideas or materials. Bloom suggests that the participant is asked to provide judgments on consistency and accuracy as well as an overall valuation. The participant should be asked to cite the specific points that support their judgment.
    - Some examples of programming tasks at this level are: to assess the correction of a program; to evaluate its efficiency, or to judge the appropriateness of design techniques: data structures and algorithms. During the working with

automatic graders, these tasks are not specifically assigned to students, but they must know them and put them into practice, because automatic graders test correct running and performance within time and space limits. The latter requirement often demands the student to entirely redesign a solution to be accepted by the judge.

In this level, when the student's proficiency increases, the analysis of cost gains importance and even it is the basic reason for applying advanced techniques as dynamic programming, backtracking, branch and bound, and so on. Prior levels include them through the features of testing for correction and for efficiency, setting upper bounds to them.

4. Criteria for the Design

Problems in most contests share the same schema (description of the problem, input format, output format, sample input, and sample output), useful for automatic testing. But these problems are usually hard to solve by current, regular students (Fitzgerald, 1996). Repositories currently available of these problems are not classified under pedagogical criteria, although there are proposals to group these large collections into specific categories, e.g. attending to the algorithmic technique required (Halim, 2008).

We propose to also provide meta-information to assignments, in particular with educational information. These meta-data may be used to classify and show these problems to students. In a first approach, meta-information may be organized in a table like this:

Problem	Basics	Example	Bloom	Difficulty	Time
…	…	…	…	…	…

Table 1. Meta-data table for design.

The column titled "Problem" names the problem with a short title, hyperlinked to a wording in a repository.

The "Basics" column specifies concepts practised to solve the problem, or subject of study, course or programming language or paradigm that the student must use to solve this problem.

The "Example" column relates to a simple problem. It must be fully solved and carefully presented. Alternatively, this column can refer to another problem in this same collection. Elsewhere, it has been advocated the convenience to give programs to the students so that they acquire the habit of reading as a way to learn programming (like in a children reading course) and prior to writing programs (Shepard *et al.*, 2001; VanDeGrift, 2005). In addition, reusing is fostered by referring to previously designed programs.

The "Bloom" column indicates the level that the student should achieve to solve the problem. Although there exist many studies on the application of Bloom's taxonomy, there still are ambiguities and misconceptions that are not easily resoluble (Hernán-Losada *et al.*, 2004). The level depends strongly of context. It can be more easily determined by using the concepts included in "Basics" and on the task demanded. For example, computing the factorial of a natural number with its recursive definition, when students are novices in

recursion, may fit the synthesis level. The same problem may fit the application level if the students had developed the program to calculate the triangular numbers in recursive form.
The "Difficulty" degree ranges from 1 to 5, roughly considering that the number of students unable to solve it ranges, respectively, in the percentile table 10 25 50 75 90. Note the difficulty of stating in general this scale more accurately, because it depends on the students' background, whether they are majors or not, the sample provided as a reference, and so on. Nevertheless, this column is currently under empirical study.
The limit "Time" is currently defined according to our personal experience. There are obstacles to try to make it concrete. Both variables, difficulty and time, are inversely related, "Time" is generously stated, meaning that these problems are stated as practices. These data are useful in automatic graders.

5. Discussion

The main rationale of our proposal is pedagogical: it is grounded in the first-test approach and Bloom's taxonomy. The underlying is that just grading or testing is not enough for effective learning of programming.
A review of the literature on the application of Bloom's taxonomy to learning programming reveals a number of ambiguities and contradictions among different authors (Fuller *et al.*, 2007; Hernán-Losada *et al.*, 2004), but it is a widely accepted framework and we consider it very useful to assess student's knowledge and to classify assignments. Lister and Leaney (2003) encourage teachers to design assessment according to the cognitive Bloom's levels.
McCracken *et al.* (2001) found in their study that the programming skills of first year CS students were much lower than expected. In fact, most students could not complete the required programming task in the given time. If programming assignments are designed according to different cognitive levels and they are shown in the correct sequence (in grade of increasing difficulty) we can assist more effectively in the learning process of students. In addition, his or her motivation is increased because of several reasons: the problems themselves, the possibility of submitting them and obtaining an immediate answer with feedback, and the possibility of celebrating small contests in the classroom.
Admittedly, our approach does not try to be exhaustive: an actual course needs a variety of other exercises, some of which cannot be assessed automatically. For instance, theoretical issues must be addressed, examples of a good programming practice must be shown, and reading programs by design tests must be fostered.

6. Conclusion and future works

Some authors (Daly & Waldron, 2002) claim that programming exercises are activities at the application level of Bloom's taxonomy. Indeed, this is the ultimate goal, but we claim that it would be desirable to take the taxonomy into account from the start. And it is possible to do this from the comprehension level without renouncing the test-first approach, and automatic grading. Applying these criteria allows a pedagogical system managed for the instructor to show problems in the adequate order to smooth the learning curve.
A more specific conclusion is that an assignment may only be placed at a given level of Bloom's taxonomy in context, because it depends on the information supplied in the

statement, the temporal proximity of the applied theory or even the similarity to other examples.

We have observed that students' motivation increases with this approach because they obtain immediate feedback and because of the possibility of celebrating small contests in the classroom itself. Finally, reading code is an under-practiced activity (Lister, 2004; VanDeGrift, 2005) and different sorts of the exercises proposed foster to read code, especially those placed at comprehension level.

Finally, our approach is not intended to cover all activities of the software process. For instance, development tests, graphical user interfaces, Internet, interactive applications, database design, and many more. Other have addressed these issues, and our approach is intended to be a complimentary one, especially suitable to distance education, but not in an exclusive manner.

Future works include the development of a collection of problems with these characteristics to be used in the teaching of programming, as well as the implementation of capable system to display the exercises of mode fitted to the pupil's needs

7. Acknowledgements

Research partially supported by the Spanish MEC projects TIN2006-15578-C02-01 and TIN2008-04103/TSI.

8. References

Ala-Mutka, K.M.: A survey of automated assessment approaches for programming assignments, *Computer Science Education*, 15(2), June 2005, pp. 83-102.

Anderson, L. W. & Krathwohl, D. A.: A Taxonomy for Learning, Teaching, and Assessing: A Revision of Bloom's Taxonomy of Educational Objectives. *Addison-Wesley*, 2001.

Beck, K.: *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, 1999.

Beck, K.: Aim, fire (test-first coding). *IEEE Software*, 18(5), Sept.-Oct. 2001, pp. 87-89.

Bloom, B.; Furst, E.; Hill, W. & Krathwohl, D.R.: Taxonomy of Educational Objectives: Handbook I, The Cognitive Domain, *Addison-Wesley*, 1956.

Daly, C. & Waldron, J.; Introductory Programming, Problem Solving and Computer Assisted Assessment, *Proceedings of the 6th Computer-Assisted Assessment Conference (CAA)*, Loughborough, 2002, pp. 93-104.

Douce, C.; Livingstone, D. & Orwell, J.: Automatic test-based assessment of programming: A review, *ACM Journal of Educational Resources in Computing*, v.5 n.3 4-es, 2005, pp.1-13

Edwards, S.H., Börstler, J., Cassel, L.N., Hall, M.S., Hollingsworth, J., Developing a common format for sharing programming assignments, *ACM SIGCSE Bulletin*, 40(4), December 2008, pp. 167-182.

Fitzgerald, S. & Hines, M.: 1996. The computer science fair: an alternative to the computer programming contest. In *Proceedings of the Twenty-Seventh SIGCSE Technical Symposium on Computer Science Education*, K.J. Klee (ed.) SIGCSE '96. ACM, New York, NY, 368-372. February 15 - 17, 1996.

Fuller, U.; Johnson, C.G.; Ahoniemi, T.; Cukierman, D.; Hernán-Losada, I.; Jackova, J.; Lahtien E.; Lewis, T.L.; McGee Thompson, D.; Riedesel, C.; Thompson, E; Developing a Computer Science-specific learning taxonomy. *ACM SIGCSE Bulletin* 34(4), pp. 152-170, ACM Press, New York, 2007.

Gregorio-Rodríguez, C., Llana-Díaz, L., Martínez-Unanue, R., Palao-Constanza, P., Pareja-Flores, C., Velázquez-Iturbide J.Á.: eXercita: Automatic Web publishing of programming exercises. In *Proc. 6th Annual Conference on Innovation and Technology in Computer Science Education*, ACM Press, pp. 161-164, 2001.

Halim S., Methods to solve, http://www.comp.nus.edu.sg/~stevenha/programming/acmoj.html, and ended by .../ problem_category.html. Last accessed Nov. 2008.

Hernán-Losada, I., Lázaro-Carrascosa, C.A. & Velázquez-Iturbide, J.A.: On the use of Bloom's taxonomy as a basis to design educational software on programming. In *Proc. World Conference on Engineering and Technology Education*, pp. 351-355, 2004.

Hunt, F., Moch, J., Nevison, C., Rodger, S., and Zelenski, J., How to develop and grade an exam for 20,000 students (or maybe just 200 or 20). In *Proc. SIGCSE Technical Symposium on Computer Science Education*, ACM Press, pp. 285-286, 2002.

Lister, R. & Leaney, J.: First Year Programming: Let all the flowers Bloom, in T. Greening & R. Lister (eds.), *Computing Education 2003 Fifth Australasian Computing Education Conference*, pp. 221-230, 2003.

Lister, R., Adams, E. S., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M., McCartney, R., Moström, J. E., Sanders, K., Seppälä, O., Simon, B., and Thomas, L. 2004. A multi-national study of reading and tracing skills in novice programmers. In *Working Group Reports From ITiCSE on innovation and Technology in Computer Science Education*, pp. 119--15.

McCracken, M.; Almstrum, V.; Diaz, D.; Guzdial, M.; Hagan, D.; Kolikant, Y.; Laxer, G.; Thomas, L.; Utting, I. & Wilusz T.: A multi-national, multi-institutional study of assessment of programming skills of first-year CS students, *ACM SIGCSE Bulletin*, 33(4), ACM Press 2001 pp. 125 – 180.

Pareja-Flores, C. & Velázquez-Iturbide, J.Á.: Local vs. comprehensive assignments: two complementary approaches, *ACM SIGCSE Bulletin*, 32(4), December 2000, pp. 48-52.

Shepard, T.; Lamb M. & Kelly, D.: More testing should be taught. *Communications of the ACM*, 44(6): 103-108, June 2001.

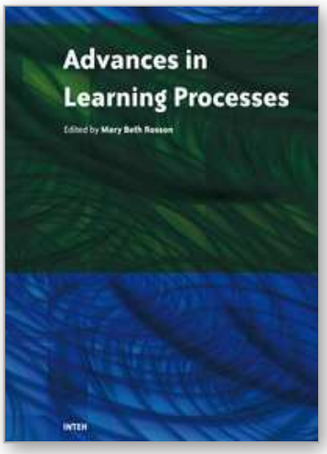
Shilov, N. V., & Yi, K. Engaging students with theory through ACM collegiate programming contest. *Communications of the ACM*, 45(9), September 2002, pp. 98–101.

Szuecs, L.: My favorite programming contest problems. *Journal of Computing Sciences in Colleges*, 17(1) , 2001, pp. 225–232.

UVA, Problem set archive: http://uva.onlinejudge.org/ Last accessed April 2009

VanDeGrift, Tammy, "Reading Before Writing: Can Students Read and Understand Code and Documentation?," *ACM SIGCSE'05*, February 2005. pp 1-4

XP, Extreme Programming. A gentle introduction, http://www.extremeprogramming.org. Last accessed Feb. 2006.



Advances in Learning Processes

Edited by Mary Beth Rosson

ISBN 978-953-7619-56-5
Hard cover, 284 pages
Publisher InTech
Published online 01, January, 2010
Published in print edition January, 2010

Readers will find several papers that address high-level issues in the use of technology in education, for example architecture and design frameworks for building online education materials or tools. Several other chapters report novel approaches to intelligent tutors or adaptive systems in educational settings. A number of chapters consider many roles for social computing in education, from simple computer-mediated communication support to more extensive community-building frameworks and tools. Finally, several chapters report state-of-the-art results in tools that can be used to assist educators in critical tasks such as content presentation and grading.

InTech Europe
University Campus STeP Ri
Slavka Krautzeka 83/A
51000 Rijeka, Croatia
Phone: +385 (51) 770 447
Fax: +385 (51) 686 166
www.intechopen.com

InTech China
Unit 405, Office Block, Hotel Equatorial Shanghai
No.65, Yan An Road (West), Shanghai, 200040, China
中国上海市延安西路65号上海国际贵都大饭店办公楼405单元
Phone: +86-21-62489820
Fax: +86-21-62489821

IntechOpen